UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 672

ECRINS
UN LABORATOIRE DE PREUVE POUR LES CALCULS DE PROCESSUS

Eric MADELAINE
Robert de SIMONE

Mai 1987

RR N° 672

ECRINS
Un laboratoire de preuve pour les calculs de processus
A proof laboratory for process calculi.

Eric Madelaine
Robert de Simone

I.N.R.I.A *Sophia Antipolis*
06560 VALBONNE FRANCE

Résumé.

Le système ECRINS est destiné à la manipulation de calculs algébriques de processus (parallèles communicants). Par cette terminologie nous voulons signifier tous les calculs cousins de *CCS*, de R. Milner, y compris de nouveaux calculs que l'on peut introduire incrémentalement dans ECRINS. Le principal instrument de définition est la *règle de réécriture conditionnelle*, grâce à laquelle on présente la sémantique opérationnelle de chaque opérateur du calcul proposé. Nous montrons ici comment ECRINS manipule de telles règles, qui sont son type de données principal, en les composant pour obtenir des "spécifications" d'expressions du calcul. Les calculs sont paramétrés par une structure d'*actions*, et une large partie spécifique d'ECRINS est dédiée à la manipulation de relations entre ces actions. Ces relations sont engendrées par des prédicats figurant dans les règles. Elles sont susceptibles d'être manipulées telles quelles, comme des prédicats "logiques" traditionnels, mais aussi dans le cas des structures d'actions "classiques" rencontrées à ce jour –ensembles finis avec inverses de *CCS*, monoïdes commutatifs de *SCCS*– par des implémentations simples, qui induisent des résultats de décidabilité. Ces aspects nous permettent en particulier de décider automatiquement de la bisimulation forte dans *CCS* et les calculs analogues sur des expressions –avec variables libres– non récursives. ECRINS est actuellement en cours de développement, en Le_Lisp sur SM90.

Abstract

The ECRINS system is dedicated to process algebras manipulation. Under this generic name we place all calculi descending from early *CCS*, by R. Milner. This includes new calculi, which the user may incrementally introduce. Main tool of definition is the *conditional rewrite rule* formalism, through which all operators are semantically presented. Such rules are ECRINS's main data type. We show here how it combines them to obtain *specifications* of expressions from a calculus. These calculi are parameterized through an *actions* structure, and so a large specific ECRINS's activity consists in dealing with it, notably in manipulating relations to hold in between actions. These relations may be dealt with formally, as traditional logical predicates. But in the case of so far prominent actions structures –finite sets with inverses or commutative monoids– they may be implemented in natural models to provide better decision results. For instance we prove to be able to decide automatically of strong bisimulation in between non-recursive open expressions –*with* free process variables– on *CCS*-like languages. ECRINS is currently being developped on SM90 in Le_Lisp.

1. Introduction.

A la suite des travaux de R. Milner sur *CCS* puis *SCCS* [Mi0,Mi1], les calculs algébriques de processus se sont révélés un formalisme de modélisation du parallélisme et de la communication au succès sans cesse croissant [Au&Bo]. Ceci est certainement dû au fait qu'ils présentent un dosage subtil de syntaxe et de sémantique, ce qui leur permet de conserver des fondements naturels (interprétation directe en systèmes de transition, bisimulations) tout en offrant la possibilité de structurer la solution de chaque problème en un système modulaire où apparaissent clairement les sous-composants (on peut "programmer" sa solution). La structure algébrique des calculs de processus, de pair avec leur méthode structurelle de définition sémantique, en faisait un objet qu'il était tentant d'implémenter en machine, pour y mener des preuves d'équivalence, des validations, des recherches de comportements. L'interêt est qu'ici on puisse disposer d'un cadre unifié où les spécifications s'expriment de manière homogène avec leurs réalisations: toutes sont les expressions d'une algèbre d'un *calcul de processus.*

Rappelons brièvement qu'un calcul de processus est, pour la syntaxe, une algèbre de termes, d'un type *processus.* Cette algèbre est paramétrée par une seconde algèbre, celle des *actions* et des *signaux.* Les opérateurs qui engendrent le calcul ont donc des arguments *processus* et des paramètres *action* –ou assimilés–, à partir desquels ils composent de nouveaux termes de type *processus.* On peut aussi créer des termes virtuellement infinis, par des *définitions récursives de processus.* Au plan sémantique ces termes s'interprètent dans des *systèmes de transitions,* dont les états sont des processus et les étiquettes sont des actions. Le point ici fondamental est que ces systèmes de transition sont construits de manière naturellement structurelle, progressivement, en suivant la syntaxe du terme à interpréter. Ceci se fait au moyen de *règles de réécriture conditionnelle* [Plotkin], attachées aux opérateurs, qui explicitent exactement comment construire les transitions du système global à partir de celles de ses arguments. Insistons ici sur l'unicité de ce formalisme des règles comme outil de définition: elles seront le type de données privilégié d'*Ecrins*, son objet d'études, qu'il va ensuite analyser, combiner et comparer. On parlera des transitions immédiates (*i.e.* au premier pas) alors déduites pour une expression comme de ses *comportements.* Ce sont à nouveau des règles de réécritures conditionnelles.

Au plan de la comparaison des expressions à partir de leurs comportements nous n'emploierons que la bisimulation forte [Park], adaptée au cas des termes non clos, ce qui exclut pour l'instant les extensions possibles comme l'équivalence observationnelle [Mi0], les équivalences selon des *observables* [Boudol] ou des *contextes* [Larsen].

Pour présenter le système *Ecrins* nous allons suivre la piste d'un exemple. Il s'agit de l'equivalence entre deux moyens de communications, déjà remarquée par R. Milner dans [Mi2]. D'un côté le *parallèle restreint* de *CCS*, de l'autre le *rendez-vous caché* de *TCSP*[BHR]. Nous avons pour l'occasion "affublé" *TCSP* d'une sémantique opérationnelle par règles de réécriture conditionnelle suivant le modèle de G. Plotkin, ce qui n'avait pas été explicitement fait jusqu'aux travaux de normalisation ISO du langage *Lotos* [IsoLo]. Les lecteurs peu familiers de *TCSP* trouveront une définition des opérateurs que nous allons en utiliser, écrits en format *Ecrins*, au chapitre suivant.

Le problème est donc de déterminer l'équivalence:

$$(p[s/a] \,||\, q[\bar{s}/b]) \backslash s \simeq (p[t/a] \,//\{t\}//\, q[t/b]) \,\backslash\backslash\, t$$

où les opérateurs sont notés comme suit:

- $.[\cdot/\cdot]$ pour le renommage (par exemple de a par s dans p).
- $.\,||\,.$ pour le parallèle *CCS.*
- $.\backslash\cdot$ pour la restriction *CCS.*
- $.\,//B//.$ pour le *rendez-vous* sur ensemble spécifié B en *TCSP.*
- $.\,\backslash\backslash\,\cdot$ pour le masquage *(hiding)* d'une action en *TCSP.*

Cette équivalence pourra être établie sous l'hypothèse que les actions visibles –a et b ou d'autres actions possiblement effectuées par p ou q– ne peuvent avoir des inverses, auquel cas aucune synchronisation "parasite" entre p et q ne peut se produire indûment dans $p[s/a] \,||\, q[\bar{s}/b]$. Il est aussi nécessaire de supposer que s et t sont de "nouveaux" noms d'actions, qui ne peuvent intervenir comme comportements de p et q. De telles hypothèses sont en général formulées assez implicitement dans les preuves d'équivalences "sur papier". Les formuler précisément au système *Ecrins* nécessite plus de rigueur.

La première section de ce papier décrit la partie "reconnaissance syntactique" du système. Il faut savoir reconnaitre les calculs corrects, puis les expressions correctes d'un calcul. La première obligation induit qu'on doive donner une forme particulière admise de règles de réécriture conditionnelle. Tout calcul candidat à analyse devra pouvoir tolérer une définition de cette forme.

La seconde section traite de l'évaluation, c'est à dire de l'exploitation des fameuses *règles* des opérateurs pour la construction structurelle de *spécifications* d'expressions. Comme nous considérons en ECRINS à la fois des expressions *non closes* et des termes récursifs, la situation est un peu plus complexe qu'il n'y parait. L'évaluateur prend la forme d'un transformateur de *spécifications*, elles mêmes considérées comme des ensembles de comportements. Dans le cas général cette composition doit être guidée par des tactiques, à la manière de LCF. Une bonne part du travail de l'évaluateur consiste à combiner les prédicats exprimant les relations entre actions, qui s'empilent progressivement. Si on refuse les définitions récursives dans les termes, et dans le cadre des structures d'actions implémentées actuellement, ces prédicats forment des ensembles d'équations et d'inéquations (diophantiennes homogènes affines) particuliers, dont les ensembles de solutions sont bien connus: ce sont les ensembles *semi-linéaires*, à cœfficients de puissance entiers pour les actions de *SCCS*, booléens pour celles de *CCS*. En version standard ECRINS traite ces prédicats de manière formelle, mais on peut forcer une traduction, dans le cas d'une structure d'action à la *CCS*, qui exprime exactement les ensembles solutions pré-cités. Cette traduction peut se voir orientée par certaines indications partielles, fournies par l'utilisateur, sur la nature de l'alphabet. Naturellement dans ce cas elle n'est valide que sous ces hypothèses. La traduction s'avère fournir une procédure de normalisation des prédicats en forme unique. Dans le cas de structure d'actions à la *SCCS*, la même construction est théoriquement possible [GS], mais l'égalité y est de complexité redhibitoire [GI].

La troisième et dernière partie de cet article décrit le *système de preuve* d'ECRINS, qui utilise les spécifications produites par l'évaluateur pour les comparer, les réduire, et finalement s'attacher à vérifier ou à infirmer des relations entre processus sujettes à bisimulation... Naturellement c'est ici qu'un principe de normalisation sur les prédicats d'actions prend tout son sel, car il permet d'avoir un algorithme complet pour ce problème de la bisimulation, dans le cas des expressions non récursives.

2. De l'analyse de la syntaxe

La première activité d'ECRINS consiste à vérifier que les objets qui lui sont présentés vérifient bien la syntaxe convenue dans les calculs de processus. A un premier niveau on vérifie qu'un ensemble d'opérateurs, regroupés dans un "calcul" particulier, est bien défini sémantiquement par des règles de réécriture conditionnelle du format admis. Dans un deuxième temps on vérifiera que des expressions écrites dans ce calcul homologué sont bien correctes, avant de pouvoir les traiter dans l'évaluateur. L'utilisateur dispose au "toplevel" d'ECRINS des commandes suivantes:

`analy "<fichier>"` permet de charger la définition d'un calcul, et de vérifier la cohérence des règles qui le constituent (d'après un format décrit ci-dessous).

`compile "<calcul>"` produit des fichiers contenant un *parser* et un évaluateur pour le calcul entré.

Ces fichiers seront chargés lorsque l'utilisateur déclarera qu'il veut travailler dans ce calcul, par la commande `calcul "<calcul>"`.

2.1. Les opérateurs, leurs règles

Comme exemple de règles sémantiques de réécriture conditionnelle telles qu'on peut typiquement les définir en ECRINS, voici trois opérateurs extrait de la définition de *TCSP* telle que nous l'avons présentée à ECRINS:

```
operator prefix:: Action, Process --> Process
semantics
    prefix
                    ------------------
                      a:p -- a --> p
end

                                                    %un Label est un nom
operator hiding:: Process, Label --> Process        %d'action ou de signal
```

sur les mêmes variables initialement, et utilisant par la suite les mêmes hypothèses, contiendront toujours exactement les mêmes variables. Telles quelles, ces restrictions ne sont pas actuellement vérifiées en ECRINS et leur réalisation est -provisoirement- de la responsabilité du concepteur de calcul.

Notons qu'au contraire des quatre éléments précédents, les variables $u\_p_{i_j}$ et $x\_p_{i_j}$ sont muettes et simplement positionnelles, ne caractérisant pas la règle.

Un opérateur est défini par un nombre fini de règles, qui forment sa *spécification*. Cette pluralité permet d'introduire du non-déterminisme notamment.

2.2. Les prédicats sur les actions

Nous avons volontairement omis de préciser jusqu'ici les syntaxes permises pour les entités *Pred* et *result*. C'est qu'elles dépendent essentiellement de la structure d'action, paramètre du calcul, sur laquelle elles opérent. A des univers d'actions différents correspondront des syntaxes de *Pred* et *result* différents. Dans la version courante, seule est implémentée la structure d'actions sous-jacente à *SCCS* et MEIJE. Celle de *CCS*, qui en est un sous-ensemble strict, s'y trouve naturellement immergée. Pour rappel, la structure d'actions de *SCCS* comprend:

- **des labels** (ou *noms d'actions*), qui sont toute suite de caractères alphanumériques, ainsi que "-" et "_"
- **des actions atomiques (atoms)** représentées en ECRINS par leurs **labels**.
- **des signaux inversibles** représentés en ECRINS par un **label** terminé par le symbole "!" pour l'émission, "?" pour la réception du signal.
- **une action neutre, ou invisible, ou cachée** notée par l'identificateur spécial *tau* en ECRINS.
- **un produit commutatif de simultanéité** noté simplement ".". Ce produit créé un monoïde commutatif. La notation $a\hat{}4$ –pour $a.a.a.a$– est utilisée dans ECRINS. De plus, par exemple, $s???$ est une autre notation pour $s!\hat{}-3$.
- **des morphismes d'actions pour les actions renommées.** Les morphismes sont des applications *Labels* $\longrightarrow$ *Actions* qui conservent l'action *tau* et le produit commutatif, donc engendrés par leur image sur les actions atomiques et les signaux. On n'utilise en ECRINS actuellement que des morphismes de *support fini*, où le support est l'ensemble des actions atomiques et signaux sur lesquels le morphisme en question n'est pas l'identité. On note par exemple $[s?/b, tau/a]$ le morphisme qui envoie b sur $s?$ et a sur l'action vide, sans toucher aucune autre. En toute rigueur un morphisme ne devrait pas pouvoir renommer un signal en action, mais ceci n'est pas vérifié actuellement. Par contre un morphisme n'étant correctement défini que si une action et un signal ne partage pas le même label, cette correction est vérifiée dans toute expression.
  Une action renommée est une action de la forme $u[\phi]$, où ϕ est un morphisme.

Les entités *Pred* et *result* utilisent de tels *termes d'actions*, basés sur certains générateurs:

- d'une part les variables d'actions $u\_p_{i_j}$ utilisées dans la règle.
- d'autre part les actions définies comme paramètres de l'opérateur F.
- des constantes d'actions déclarées globalement dans un calcul.

result, qui détermine l'action globale résultant de l'application de la règle aux processus arguments, est un terme d'actions au sens précédent.

Pred, le prédicat qui impose une relation entre les actions composantes, est une formule de la fermeture par intersection des prédicats de base:

$$<terme1> ?= <terme2>, \qquad <terme1> ?/ <terme2>, \qquad <terme> ?in \{< action-list >\}$$

$$\mathbf{not}(<terme1> ?= <terme2>), \mathbf{not}(<terme1> ?/ <terme2>), \mathbf{not}(<terme> ?in \{< action-list >\})$$

$$<terme1> ?\ inverse\ <terme2>$$

où

- $?=$ représente l'égalité entre les actions.
- $?/$ représente la divisibilité entre actions, et
- $?in\ \{< action-list >\}$ encode l'appartenance à un ensemble d'actions fini énuméré.

Ceci termine la présentation du format des règles dans le cas où la structure d'actions est celle de *SCCS*.

```
semantics
    hiding_tau      p -- a --> p'  &  (a ?= s)
                    ---------------------------
                       p\\s -- tau --> p'\\s
    hiding          p -- a --> p'  &  not (a ?= s)
                    --------------------------------
                          p\\s -- a --> p'\\s
end
operator rdv:: builtin
semantics
    rdv_left        p -- a -->p'  &  (not(a ?in B))
                    ---------------------------------
                      p//B//q -- a --> p'//B//q
    rdv_right       q -- b -->q'  &  (not(b ?in B))
                    ---------------------------------
                      p//B//q -- b --> p//B//q'
    rdv_both        p -- a -->p'  &  q -- b -->q'  &  ((a ?in B) et (a ?= b))
                    ------------------------------------------------------------
                                p//B//q -- a --> p'//B//q'
end
```

Ces définitions sont regroupées dans un fichier rassemblant tous les opérateurs du calcul avec leurs sémantiques, et des indications de syntaxe. Ce fichier est créé et édité hors d'ECRINS.

A l'heure actuelle la structure des actions –et ses types hérités– est fixée dans ECRINS(nous y reviendrons à la section suivante). Tout opérateur se voit muni d'un "type", comme par exemple ici `Action, Process --> Process`. Ce type décrit la forme de ses arguments `Processus` ainsi que ses paramètres `Action, Atom, Signal, Label`, et surtout leurs positions dans la syntaxe concrète. Il serait à priori possible pour les opérateurs d'un calcul d'avoir des paramètres de types plus sophistiqués, comme c'est déjà le cas pour certains opérateurs -prédéclarés- des calculs existants:

- *renaming*, qui utilise un paramètre de type `morphisme: labels` $\rightarrow$ `actions`, et
- *rendez-vous sur un ensemble*, de *TCSP*, qui lui utilise un paramètre de type `labels list`.

Il n'est pas loisible pour l'instant à l'utilisateur de créer des opérateurs usant de tels types de paramètres: les deux opérateurs précédents sont de type "*builtin*". Pour les réutiliser dans un calcul, il faut les déclarer de type "*builtin*", et redéfinir leurs règles de sémantique, en conservant leur syntaxe.

Le format admis pour les règles est le suivant, maintenant bien connu depuis *CCS*, *SCCS* et MEIJE:

$$\frac{p_{i_1} \xrightarrow{u\_p_{i_1}} x\_p_{i_1}, \cdots, p_{i_k} \xrightarrow{u\_p_{i_k}} x\_p_{i_k} \quad \& \quad Pred(u\_p_{i_1}, \cdots, u\_p_{i_k})}{F(p_1, \cdots, p_n) \xrightarrow{result(u\_p_{i_1}, \cdots, u\_p_{i_k})} T(\vec{p}, \overrightarrow{x\_p})}$$

Les quatre éléments significatifs d'une telle règle sont:

I l'ensemble des identificateurs de processus arguments –les p_{i_j}– qui agissent dans la règle.

Pred un prédicat portant sur une relation imposée entre les différentes actions des comportements de ces processus arguments.

result l'action globale engendrée par ces comportements. *Pred* peut aussi dépendre des paramètres *actions* de *F*.

T l'expression de processus résultant du comportement global. Cette expression est fondée sur les variables de processus p_{i_j} et $x\_p_{i_j}$, sachant que seules les p_{i_j} n'apparaissant pas dans *I*, ainsi que les $x\_p_{i_j}$ tels que p_{i_j} est dans *I*, peuvent y apparaitre. De plus, on impose que chaque variable de processus ne se trouve inscrite qu'au plus une fois dans *T*, et jamais dans un sous-terme récursif (qui pourrait éventuellement la dupliquer).

Ces restrictions sur *T* assurent une certaine cohérence du format des règles, en concordance avec l'intuition de progression temporelle naturelle des composants. En particulier, deux expressions portant

2.3. Les expressions du calcul.

Le fichier de déclaration d'un calcul décrit précédemment contient des informations syntaxiques suffisantes pour construire un analyseur syntaxique d'expressions finies (et closes). Par la commande `compile` le système *ECRINS* ajoute systématiquement au langage produit:

- des variables de processus (ou *processus libres*) pour construire des termes non clos.
- des systèmes de définitions récursives de processus, à l'aide de ces variables (`let rec` <*def*> `in` <*process-exp*>)
- des systèmes de définitions locales, non récursives, de processus, toujours à l'aide de ces mêmes variables (`let` <*def*> `in` <*process-exp*>)
- des systèmes de définitions locales de nouvelles actions (`local signal` <*action-name*> `in` <*process-exp*>) et (`local atom` <*action-name*> `in` <*process-exp*>). Ces définitions lient les occurrences de *action-name* dans *process-exp*. En particulier elles imposent – et c'est là leur but principal– que l'on vérifie que *action-name* n'est pas une action que pourraient effectuer les variables libres de processus éventuelles de *process-exp*. Ces définitions définissant une portée des effets de *action-name*, il faudra de plus assurer qu'à l'exécution de *process-exp*, l'action globale synthétisée ne comporte jamais une composante non nulle en *action-name*. Les deux méthodes classiques pour "étouffer" les occurrences d' actions sont la restriction et le hiding, justement les deux opérateurs de notre exemple...

Tout ceci forme un langage, pour lequel un *parser* est chargé lors de l'utilisation du calcul, déclarée par `calcul "`<*calcul*>`"`. Par la suite une commande de la forme: `parse` <*ident*>= <*process-exp*> permet d'analyser syntaxiquement un terme avant de l'évaluer. De plus, la commande `parse` vérifie qu'aucun label ne désigne à la fois un signal et une action atomique.

2.4. Les contraintes à l'alphabet

Ayant lié par la commande `parse` un terme du calcul à une variable *ECRINS*, l'utilisateur dispose des commandes suivantes qui pratiquent des tests de sémantique statique sur le terme:
`base` <*ident*>

Le rôle de `base` est de définir un alphabet "minimal" de *labels* d'actions dans lequel la preuve d'équivalence pourra être conduite correctement. Le calcul de la base s'effectue structurellement en remontant l'expression, et en récoltant tous les labels syntaxiquement présents, dans les renommages, dans les paramètres de type Action ou Label, ainsi que les labels introduits explicitement par certains opérateurs. Cette base est partagée en deux entre les actions locales à l'expression –introduites par une définition en `local` <*action*> `in` ..., et qui sont transformées en indices entiers–, et les autres.

```
@ parse x = local signal s in (p[s!/a] // q[s?/b])\s ;
Parsed x:ccs

@ base x ;
< s(1) | a b>              % 1 étant le numéro accolé à s.
```

La barre sépare les actions atomiques des signaux, et les actions locales sont suivies de leur numéro de représentation interne. Les signaux liés de x sont donc bien $s!$ et $s?$, les actions visibles étant a et b.

`sorte` (<*ident*> {`with` <*contraintes*>}.

Cette commande calcule l'ensemble des actions et signaux visibles d'une expression. La présence de variables libres de processus complique sa définition: on a besoin de spécifier quelles actions chaque variable de processus peut faire. Il s'agit la plupart du temps de leur interdire tout signal inversible, ou de leur attribuer un alphabet d'actions fini. Ces contraintes seront également utilisées pour évaluer les comportements d'un term, ou pour faire une preuve de bisimulation.
<*contraintes*> est une liste de déclarations de la forme

`sorte` <*vlist*> `is atoms` <*spec*> `signals` <*spec*>,

où *vlist* est une liste de variables de processus, et *spec* est ou bien une liste de `labels`, ou bien un des deux mots-clés `none` pour la liste vide, et `any` pour un ensemble indéfini. Si l'argument <*contraintes*> n'est pas

fourni, alors on calcule dans un environnement indéfini sur les sortes des variables de processus, ce qui correspond à `sorte <vlist> is atoms any signals any`. Tout comportement calculé dans cet environnement sera universellement vrai. Les autres devront porter mention des circonstances qui les valident.

Nous pouvons maintenant décrire le calcul de la `sorte`: les variables libres de processus ont à priori pour `sorte` l'intersection de la **base** de l'expression, dont on supprime les actions locales que les processus libres ne peuvent faire par hypothèse, avec la *sorte* qu'on leur a éventuellement déclarée par *contraintes*. On y ajoute pour chaque variable libre, et sauf si sa sorte est explicitement énumérée, une *action externe*, inversible ou non selon la contrainte existante, et notée par convention u_p (pour un processus p). Ensuite une `sorte` est définie structurellement pour chaque sous-expression, à partir de l'expression de l'action *result*, telle qu'on la trouve dans toutes les règles de réécriture concernées. Il faut ici être attentif à **soustraire** de la `sorte` toute action telle que dans cette règle justement elle soit:

- ou bien exclue de la sorte de *result*, en étant renommée en autre chose dans le terme de cette action (ce qui est le cas dans `hiding`).
- ou bien exclue de la sorte de *result* car dans le predicat apparait une formule de base de la forme `not <action>?/result`.
  Une expression ayant une déclaration d'action locale (`local ...`) comme opérateur de tête jouera un rôle particulier, car elle fera échouer le calcul de la `sorte` si l'action qu'elle lie figure parmi celles admises de son processus fils. Ceci est obligatoire car sinon l'action en question pourrait virtuellement être propagée vers l'extérieur, ce qui est contraire à l'idée d'action locale...

```
@ sorte x with  sorte {p,q} is atoms any signals none  ;
{a b u_p u_q}
```

3. L'évaluateur

La commande `compile <calcul>`, on l'a vu, crée un parser à partir de la présentation du calcul. Mais elle en extrait aussi un *évaluateur*. *Evaluer* une expression, c'est calculer l'ensemble des comportements qu'elle peut avoir *en un pas élémentaire*. Le résultat de l'évaluation est une *spécif*, un ensemble de règles de réécriture conditionnelle.

```
@ eval ccs-term ;
Specif of (local signal s in (p[s!/a] || q[s?/b])\s)
    {p} {not s ?/ u_p<s!/a>} -- u_p<s!/a> --> local signal s in (x_p[s!/a] || q[s?/b])\s
    {q} {not s ?/ u_q<s?/b>} -- u_q<s?/b> --> local signal s in (p[s/a] || x_q[s?/b])\s
    {p,q} {u_q<s?/b> ?inverse u_p<s!/a>} -- tau -->
          local signal s in (x_p[s!/a] || x_q[s?/b])\s
```

Comment lire le résultat de cette commande ?
Chaque règle de la *spécif* contient d'abord l'ensemble des processus libres qui y évoluent, puis une condition d'applicabilité (un *prédicat*) portant sur les *actions formelles* de ces processus libres, et enfin le comportement obtenu, sous la forme d'une transition. Par convention, le système note u_p l'action formelle du processus libre p, et x_p le processus résultant correspondant. Par exemple
"`{p} {not s ?/ u_p<s!/a>} -- u_p<s!/a> --> local signal s in (x_p[s!/a] || q[s?/b])\s` " n'est qu'une notation plus compacte de:

$$\frac{p \xrightarrow{u\_p} x\_p, \quad \texttt{not}\ (s\ ?/u\_p\texttt{<}s!/a\texttt{>})}{\texttt{ccs-term} \xrightarrow{u\_p\texttt{<}s!/a\texttt{>}} \texttt{local signal}\ s\ \texttt{in}\ (x\_p[s!/a]\,||\,q[s?/b])\backslash s}$$

Les règles figurant dans les spécifs sont en fait plus générales que celles admises pour les opérateurs, pour permettre d'exprimer les comportements d'expressions non-linéaires dans un processus libre. Par exemple, $p\,//\,p$ (parallèle MEIjE) a un comportement: "`{p} {} --u1_p.u2_p-->` $x1\_p\,//\,x2\_p$".

Notre exemple précédent a un ensemble de comportements décrit par un nombre fini de règles. Ceci découle de l'application à un terme fini d'un nombre fini de règles structurelles, et pourrait être étendu aux termes récursifs "dans le temps". Mais la syntaxe d'*E*CRINS permet aussi les définitions récursives non-gardées, immédiates, que l'on peut utiliser pour décrire des processus infinis (horloges, file non bornée, tas, réseaux systoliques).

Considérons par exemple en MEIJE l'horloge:

```
@ parse H_a+ = let rec x = a:0//x in x;
Parsed H_a+: meije;
```

Une description finie de sa spécif (infinie) pourrait être: $\dfrac{n \in \mathbb{N}}{H\_a^+ \text{--} a^n \text{-->} H\_a^+}$

Il est clair cependant que nous ne pouvons pas obtenir ce résultat par simple énumération des règles de la composition parallèle. Il faudrait disposer d'un principe de récurrence sur la forme des arbres de preuve. Par contre, `eval` peut nous donner simplement des résultats partiels sur cette spécif: pour tout entier positif n fixé, nous pouvons guider `eval` et prouver que le comportement -- a^n --> H_a^+ est un comportement possible de notre horloge. Il nous faut pour cela entrer plus en détail dans le fonctionnement de *eval*.

L'évaluation d'une expression se fait en deux passes: d'abord un parcours descendant de l'expression pour créer un *arbre de décomposition*, puis un parcours ascendant de cet arbre pour synthétiser les ensembles de règles de comportement. La création de l'arbre de décomposition est contrôlable par l'utilisateur au moyen de *tactiques*. Pour chaque sous-terme considéré, il faut choisir lesquelles parmi les règles de l'opérateur de tête on veut appliquer. Ceci se fait par des tactiques élémentaires correspondant à chacune des règles de réécriture (et désignée par le même nom), ou par des tactiques permettant d'en sélectionner un sous-ensemble (*None, First, All,...*). Le système construit alors un nœud de l'arbre de décomposition dont les fils sont les sous-termes mis en jeu par la ou les règles sélectionnées.

```
@ parse x = p//{a}//q;
Parsed x : plotos;

@ eval x;                          % par défaut on applique All à chaque opérateur
Specif of (p//{a}//q)
    {p}{not(u_p ?in {a})} -- u_p --> x_p//{a}//q
    {q}{not(u_q ?in {a})} -- u_q --> p//{a}//x_q
    {p,q}{(u_p ?in {a}) ∧ (u_q ? = u_p)} -- u_p --> x_p//{a}//x_q
```

On peut spécifier quelle tactique appliquer à chaque nœud, en donnant un second argument à `eval`. Essayons ici `Interact`, qui demande une tactique à l'utilisateur, chaque fois qu'elle en a besoin:

```
@ eval x interact;                 % La tactique est un argument supplémentaire.
Ω (x_p//{a}//x_q)                  % Le système affiche le sous-terme courant.
tactic? rendez-vous-both;          % L'utilisateur répond par le nom de la règle choisie
Specif of (x_p//{a}//x_q)
    {p,q}{u_p?in{a} ∧ u_q = u_p} -- u_p --> x_p//{a}//x_q
```

Ces tactiques peuvent être combinées, dans le style de [LCF], à l'aide de combinateurs (`Then`, `Then1`, `Orelse`, `Repeat`...), pour former des programmes complexes. On obtient ainsi une tactique pour l'un des comportements de H_a^+, utilisant comme tactiques de base celles qui sont engendrées à partir des règles du parallèle MEIJE par la commande compile:

```
@ set get4 =
    (Repeatn 3
        (parallel-both Thenl {sequence, Idtac}))
    Then parallel-left Then sequence ;
@ eval H.a+ get4;
Specif of (H.a+)
    {} {} --a4--> 0//0//0//0//H.a+
```

Sur un terme virtuellement infini commme $H.a^+$ la tactique `All` va boucler. Nous laissons à l'utilisateur la responsabilité d'éviter ce cas. Dans la suite du papier, nous ne considèrerons plus que des termes ayant une *spécif* finie.

La passe de synthèse des comportements est automatique: en remontant depuis les feuilles de l'arbre de décomposition on calcule la spécif relative à chaque sous-arbre. L'arbre de décomposition contient dans chacun de ses nœuds la *fonction de preuve* nécessaire à calculer les comportements du sous-terme relatif à ce nœud, en fonction des comportements de ses fils. En particulier, pour chaque comportement synthétisé on calcule un prédicat *Pred* sur les actions formelles des processus libres du sous-terme, formé par conjonction des prédicats issus des fils, et de celui, instancié, de la règle de l'opérateur. L'action résultante *result* est une instanciation correspondante de l'expression figurant dans la règle de l'opérateur. L'action globale obtenue est simplifiée, ainsi que celles qui peuvent figurer dans les prédicats. En présence du produit de simultanéité, ceci permet de prendre en compte sa commutativité et son associativité, en triant les composants par type (signaux liés, actions atomiques, actions formelles), et par ordre alphabétique. Par exemple, "a^3.s?.u_p.a", où s est lié, est simplifié en "s?.a^4.u_p".

Mimons par exemple quelques pas de l'évaluation de *local signal* s *in* $(p[s!/a] \,||\, q[s?/b])\backslash s$

- p a une seule règle: $\{p\}\{\}$ --u_p-->x_p
- La règle de l'opérateur *renaming* change seulement l'action résultante:
  $p[s!/a]$ a une seule règle: $\{p\}\{\}$ --$u\_p < s!/a >$ -->$x\_p[s!/a]$
  $q[s?/b]$, de la même façon: $\{q\}\{\}$ --$u\_q < s?/b >$ -->$x\_q[s?/b]$
  - $p[s!/a] \,||\, q[s?/b]$, au nœud correspondant à la règle *interleave-both*, aura la règle:
  $\{p, q\}$ $\{u\_q<s?/b>$ *?inverse* $u\_p<s!/a>\}$ -- *tau* --> $x\_p[s!/a] \,||\, x\_q[s?/b]$
- Cette même règle, transformée par la règle *restriction*, devient:
  $\{p, q\}$ $\{(u\_q<s?/b>$ *?inverse* $u\_p<s!/a>) \wedge$ **not** $(s!\ ?/\ tau)\}$ -- *tau* --> $(x\_p[s!/a] \,||\, x\_q[s?/b])\backslash s$
- l'opérateur *local signal* s *in* laisse la règle inchangée, puisqu'on a vérifié statiquement que s était bien local.

Seul le prédicat "{not (s! ?/ tau)}", directement réductible en *vrai* par le système, n'apparait pas dans le résultat.

La partie gauche de notre formule s'évalue de manière similaire:

```
@ eval tcsp-term ;
Specif of (local atom t in (p[t/a] //{t} // q[t/b]) \\ t)

    {p} {(u_p<t/a> = t) ∧ not(u_p<t/a> ?in {t})}
          -- tau --> local atom t in (x_p[t/a] //{t} // q[t/b]) \\ t

    {q} {(u_q<t/b> = t) ∧ not(u_q<t/b> ?in {t})}
          -- tau --> local atom t in (p[t/a] //{t} // x_q[t/b]) \\ t

    {p} {(u_p<t/a> = t) ∧ ((u_p<t/a> = u_q<t/b>) ∧ (u_p<t/a> ?in {t})}
          -- tau --> local atom t in (x_p[t/a] //{t} // x_q[t/b]) \\ t

    {p} {(u_p<t/a> ≠ t) ∧ not(u_p<t/a> ?in {t})}
          -- u_p<t/a> --> local atom t in (x_p[t/a] //{t} // q[t/b]) \\ t

    {q} {(u_q<t/b> ≠ t) ∧ not(u_q<t/b> ?in {t})}
          -- u_q<t/b> --> local atom t in (p[t/a] //{t} // x_q[t/b]) \\ t

    {p} {(u_p<t/a> ≠ t) ∧ ((u_p<t/a> = u_q<t/b>) ∧ (u_p<t/a> ?in {t})}
          -- u_p<t/a> --> local atom t in (x_p[t/a] //{t} // x_q[t/b]) \\ t
```

On remarque combien il est difficile d'exploiter ce résultat. Les prédicats des première, seconde et sixième règles se réduisent à *faux*. Il reste trois règles correspondant à celles de *ccs−term*, avec les mêmes processus libres qui évoluent, mais là encore, les prédicats sont difficiles à comparer formellement.

Une *simplification logique* des prédicats, même si elle ne pourrait être complète, devrait être ici envisagée; dans certains cas elle saurait même suffire (en général en l'absence d'opérateurs de synchronisation dans les termes). Le point le plus important est de pouvoir détecter les prédicats équivalents à *faux*, pour pouvoir supprimer la régle correspondante, et faire maigrir les spécifications.

Cette *simplification logique* n'est pas implantée actuellement, mais est prévue dans les développements prochains.

Une autre approche consiste en une *traduction sémantique*: dans le cas des structures d'actions de *CCS/TCSP* et de *SCCS*/MEIJE, on sait encoder les prédicats précédents par une représentation simple dans des modèles implémentant ces structures. Cette traduction, inspirée de l'algèbre linéaire, revient à écrire des matrices (LEM, pour "*Linear Enumeration Matrixes*") dont les lignes sont les variables formelles d'actions (*result* et les u_{i_k} correspondant aux processus libres actifs dans la règle), et dont les colonnes sont les noms d'actions et de signaux générateurs de l'alphabet "nécessaire", la base (cf section 1.4), sur laquelle les actions se décomposent. Les cœfficients sont alors obtenus en résolvant (par unification) les équations contenues dans les prédicats. Tout ceci implique de pouvoir fournir à eval les *contraintes* à la base. La commande se généralise donc naturellement en `eval <process-exp> with <contraintes>`.

Dans le cas de *SCCS* on obtient comme cœfficients des ensembles semi-linéaires formés sur des variables de $\mathbb{N}$. Prenons par exemple en MEIJE le terme `''local signal s in (p[s!/a] // q[s!/a] // r[s?/b])\s''`. L'une des règles de sa spécif correspond à une communication entre les trois processus, avec le prédicat: not (*s ?/ u_p<s!/a>.u_q<s!/a>.u_r<s?/a>*).

Si la seule action externe est *a*, la solution de cette équation est l'ensemble semi-linéaire:
$< u\_p, u\_q, u\_r >=< a, tau, a >^* . < tau, a, a >^*$. Si on prend en compte le fait que *p*, *q* ou *r* peuvent effectuer d'autres actions non inversibles, que nous représenterons ici par une action externe spécifiée par l'identité du processus qui l'exécute, comme *ext_p* par exemple, alors il faut rajouter àla solution le facteur $< ext\_p, tau, tau > + < tau, ext\_q, tau > + < tau, tau, ext\_r > + < tau, tau, tau >$. On obtient les matrices précédemment décrites en exprimant chaque action (*u_p, u_q, et u_r*) par ses exposants sur les coordonnées de la base. Une puissance quelconque sera représentée par une variable entière libre. L'égalité sur cette structure est décidable [GS], mais de très grande complexité. Nous n'avons pas encore implanté ce mécanisme, car il semble que des méthodes interactives de preuve seront ici nécessaires.

Pour le cas de *CCS* on obtient plus simplement des expressions booléennes formées à partir de variables booléennes. En l'absence de produit de simultanéité, une seule au plus des variables booléennes peut être non nulle, et on peut énumérer les vecteurs-colonnes solutions. Une telle liste de vecteurs est dénotée ici une BEA (*"Boolean Enumeration Arrays"*). C'est la solution *in extenso* du prédicat concerné. Sur cette structure toutes les questions sont immédiatement décidables par simple comparaison.

Revenons un instant sur la base. Nous avons a priori besoin en plus d'une action externe par processus libre, mais comme on ne peut pas imposer de contraintes sur les actions externes (il faudrait les mentionner explicitement, donc elles seraient dans la base), leur seule possibilité est de remonter intactes dans l'action résultat. Ceci nous permet de nous limiter dans les structure d'actions de *CCS* à un seul et unique signal voire parfois une seule action externe pour tout le terme, puisqu'il n'y a pas ici de représentation de la simultanéité.

Reprenant l'exemple de `local atom` t `in` $(p[t/a] //\{t\}// q[t/b]) \setminus\setminus t$ avec la contrainte `{sorte {p,q} is atoms any signals none }` on obtient le tableau de correspondance suivant:

numéro	0	1	2	3	4
composant	*tau*	*t*	*a*	*b*	*externe*

Les signaux inversibles, lorsqu'il y en a, sont codé par un entier n pour l'émission, et son inverse $-n$ pour la reception.

Le prédicat "`not` $(u_p\langle t/a\rangle ? = t)$ $\wedge$ `not` $(u_p\langle t/a\rangle$ `?in` $\{t\})$" a pour traduction la BEA:

u_p	0 3 4
res	0 3 4

Cela se lit par colonne: $u\_p = tau \wedge res = tau$ ou $u\_p = b \wedge res = b$ ou $u\_p = externe \wedge res = u\_p$.

Le format des spécifs en est changé: pour chaque règle, le prédicat est remplacé par l'affichage ("en dur", avec les indices!) de la BEA correspondante.

On passe dans le mode où les prédicats sont traduits en BEA par la commande `bea`. Pour revenir au mode syntaxique, on utilise la commande `logic`. Noter qu'en mode `logic` les *contraintes* sont ignorées.

```
@ bea;
Module BEA Loaded.

@ eval x with  sorte {p,q} is atoms any signals none   ;
Base = <0 = tau  1 = t  2 = a  3 = b  4 = ext>
Specif of (local atom t in (p[t/a] //{t}// q[t/b]) \\ t)
Behaviours:
    {<Bea u_p |2
          u_q |3
          res |0>}--tau--> local atom t in (x_p[t/a] //{t}// x_q[t/b]) \\ t
    {<Bea u_p |0 3 4
          res |0 3 4>}--u_p--> local atom t in (x_p[t/a] //{t}// q[t/b]) \\ t
    {<Bea u_p |0 2 4
          res |0 2 4>}--u_q--> local atom t in (p[t/a] //{t}// x_q[t/b]) \\ t
```

Le codage sémantique en BEA nous a permis d'obtenir une forme utilisable des spécifs de nos termes. Nous pouvons maintenant nous intéresser à la preuve de leur équivalence.

4. Les preuves de bisimulation

Nous ne traiterons ici que des bisimulations "fortes", par opposition aux bisimulations *modulo* un critère d'observation [Boudol] et omettrons l'adjectif "forte" dans la suite.

La définition classique de bisimulation utilise une quantification universelle sur les comportements possibles des processus. En ce sens, c'est une définition *extensionnelle*. Lorsqu'on est en présence de termes contenant des processus libres, la relation à considérer est celle où les processus libres sont remplacés par

toutes leurs instanciations possibles. Pour pouvoir traiter (mécaniquement) ce cas [RdS] a introduit la notion de *FH-bisimulation*. Elle utilise une mise en correspondance non pas des comportements des processus, mais des règles de leurs spécifs. Cette définition est *intensionnelle* dans le sens où elle est fondée sur un codage syntaxique (fini) de l'ensemble des comportements.

4.1. Définition

Une relation d'équivalence $\Re$ entre processus est une *FH-bisimulation forte* ssi:

$\forall p, q$ tels que $p \,\Re\, q$, $\forall r$ règle telle que $r \in \mathbf{Specif}(p)$, $r = \dfrac{I, Pred}{p \to p'}$

$$\exists s,\ s \subseteq \mathbf{Specif}(q),\ s = \{s_j\}_j,\quad s_j = \frac{J, Pred_j}{q \to q'_j}$$

tels que:

$$\begin{cases} I = J \\ \forall j,\ (p', q'_j) \in \Re \end{cases} \quad \& \quad Pred \subseteq \bigcup_j Pred_j$$

Ici donc, chaque règle de comportement d'une expression doit être recouverte par éventuellement plusieurs règles de l'autre. Comme précédemment, l'union de toutes les FH-bisimulations est une FH-bisimulation, que nous appelons *FH-congruence* et notons $\underset{\text{FH}}{\cong}$.

Propriété [deS]: $\underset{\text{FH}}{\cong} \subset \cong$

Si l'on se place dans un cadre de calculs non typés, ou l'alphabet global des actions n'est pas une entité connue manipulable, la réciproque est vraie. Sinon, il existe des cas pathologiques de termes équivalents par la bisimulation forte, mais pas par la FH-bisimulation (cf. [RdS]).

Les relations que manipulent ECRINS et dont on veut prouver qu'elles sont des bisimulations sont des relations engendrées par un système fini d'équations entre processus $\{p_1 == q_1, \ldots, p_k == q_k\}$. Dans notre exemple, on n'a qu'une seule équation, on peut se passer des accolades:

```
@ parse r = local signal s in (p[s!/a] ||q[s?/b])\s ==
          local atom t in (p[t/a] //{t} // q[t/b]) \\ t;
Parsed r : ccs+tcsp equation
```

Souvent après avoir prouvé des propriétés équationnelles "essentielles" d'un calcul, ses lemmes, on voudra les utiliser pour raccourcir les preuves ultérieures. Nous appellons *simplificateur* un ensemble fini de lemmes qu'on a su orienter pour former un *système de réécriture* confluent et à terminaison finie (cf [HuetOpp]). Ces lemmes devront toujours avoir été prouvés par bisimulation. Un tel système de réécriture calcule une certaine forme normale des termes de l'algèbre des processus (Cette définition permet d'éviter de faire spécifier par l'utilisateur une stratégie de réduction ad-hoc pour chaque ensemble de lemmes, dont il faudrait vérifier la complétude).

4.2. Algorithme de FH-bisimulation.

Pour tester si une relation $\Re$ est une FH-bisimulation (modulo un simplificateur S), nous *normalisons* indépendemment les spécifs de chaque membre des équations. Cette *normalisation* comprend un quotient par la relation $\Re \bigcup S$, accompagné par un compactage de l'ensemble de règles. Ensuite les règles obtenues pour deux membres d'une équation doivent se correspondre une à une.

Considérons:

$\Re$ une relation décrite par un ensemble fini d'équations (p_i, q_i),

$\overset{\circ}{\Re}$ la relation d'équivalence obtenue en complétant $\Re$ par fermeture réflexive, symétrique, transitive et substitutive.

S un simplificateur, et $\downarrow_S$ la fonction de réécriture associée.

• $\overset{\circ}{\underset{\Re,S}{\longleftrightarrow}}$ est la relation d'équivalence entre règles définie par:

$$\frac{I_1,\ P_1}{t \to r_1} \overset{\circ}{\underset{\Re,S}{\longleftrightarrow}} \frac{I_2,\ P_2}{t \to r_2} \quad \text{ssi} \quad I_1 = I_2\ \&\ (r_1 \downarrow_S,\ r_2 \downarrow_S) \in \overset{\circ}{\Re}$$

• $Norm_{\overset{\circ}{\Re},S}$ est la fonction de normalisation des Spécifs définie comme suit:

Quotienter une spécif $Specif(t)$ par $\overset{\circ}{\underset{\Re,S}{\longleftrightarrow}}$ nous donne un ensemble de classes d'équivalence. Chaque classe est un ensemble de règles de la forme $\{\frac{I_k,\ P_k}{t \to r_k}\}$, où les I_k sont identiques, les r_k équivalents. Pour chacune de ces classes, $Norm_{\overset{\circ}{\Re},S}(Specif(t))$ forme l'union des prédicats dans la règle $\frac{I_1,\ \cup_k P_k}{t \to r_1 \downarrow_S}$.

• Tester si $\overset{\circ}{\Re}$ est une bisimulation c'est:

$\forall (p_i, q_i) \in \Re$:

i) Calculer les spécifs $S(p_i)$ et $S(q_i)$ des deux expressions.

ii) Normaliser chaque spécif: $S'(p_i) = Norm_{\overset{\circ}{\Re},S}(S(p_i)),\quad S'(q_j) = Norm_{\overset{\circ}{\Re},S}(S(q_j))$

iii) Tester si $S'(p_i)$ et $S'(q_j)$ sont en bijection, par la relation:

$$R1 = \frac{I_1,\ P_1}{p \to r_1} \leftrightarrow R2 = \frac{I_2,\ P_2}{p \to r_2} \quad \text{ssi} \quad R1 \overset{\circ}{\underset{\Re,S}{\longleftrightarrow}} R2 \ \&\ P1 \equiv P2$$

$\overset{\circ}{\Re}$ est une FH-bisimulation modulo S si et seulement si il y a bijection (pour tous les couples (p_i, q_j)).

Implémentation actuelle:

L'algorithme ci-dessus permet de décider si $\overset{\circ}{\Re}$ est une bisimulation sous deux hypothèses: Il faut être capable de décider de l'appartenance à $\overset{\circ}{\Re}$, et de l'équivalence de deux prédicats. Ceci n'est pas toujours possible, et correspond à des restrictions dans l'implémentation existantes dans ECRINS.

- Dans le cas général, tester si un couple d'expressions (contenant des variables) appartient à une relation définie par un système d'équations, c'est tester la validité d'une formule dans une théorie inductive présentée équationnellement, et c'est indécidable. Le système teste effectivement l'appartenance à la fermeture réflexive, symétrique et substitutive du système d'équations, mais non à la congruence engendrée en ajoutant la transitivité. Ceci est suffisant pour beaucoup d'applications, mais introduit une source d'incomplétude de l'algorithme.
- Dans le cas où les prédicats sont manipulés de manière formelle (en mode `logic`), on ne teste que leur identité syntactique. Ceci est une limitation importante, mais permet cependant d'obtenir des résultats pour les propriétés les plus simples des opérateurs. Par contre, si l'on utilise, dans l'algèbre d'actions de *CCS/TCSP*, le codage sémantique fini sous forme de "BEA", l'égalité des prédicats est décidable.
- Enfin, la version actuelle ne permet pas de paramétrage par un simplificateur. On peut contourner cette difficulté en introduisant explicitement les lemmes nécessaires dans le système d'équations, quitte à les reprouver maintes fois.

L'algorithme s'invoque sous ECRINS par la fonction `prove`, avec une relation en argument. Tout comme pour `eval` une *contrainte* peut aussi indicer cette commande. `prove` a trois types de résultats possibles:

- Soit pour toutes les équations, on a trouvé une bijection entre les règles. La relation (fermée par réflexivité, symétrie et substitutivité) est une FH-bisimulation. Le résultat est alors un *Théorème* qui s'écrit "⊢ `Isbisim` < *relation* >".
- Soit pour une équation, on a trouvé pour un membre une règle dont on puisse affirmer que le prédicat n'est pas équivalent à *faux*, et telle qu'aucune règle de l'autre membre n'aît le même ensemble de processus libres qui évoluent. C'est un cas où l'on peut décider que ces deux membres ne sont pas dans la bisimulation maximale, donc la relation ne peut pas être une bisimulation. Le système affiche alors le contre-exemple, et le résultat est le théorème "⊢ `Not-Isbisim` < *relation* >".
- On peut enfin ne pas savoir décider, soit parce qu'il manque un lemme, soit parce qu'en mode `logic` on ne sait pas comparer deux prédicats ... Le système alors déclare échouer, en indiquant pour quelle règle il n'a pu obtenir de correspondance.

4.3. Exemples

Exemple 1.

Terminons ici la preuve d'équivalence qui nous a servi d'exemple tout au long du papier. Elle se place bien sûr dans le monde *CCS/TCSP*, et utilise le codage des prédicats en BEA.

```
@ parse r = local signal s in (p[s!/a] || q[s?/b])\s ==
          local atom t in (p[t/a] //{t} // q[t/b]) \\ t
Parsed r : ccs+tcsp relation

@ prove r with  sorte {p,q} is atoms any signals none   ;
Base = <0 = tau  1 = s!  -1 = s?  2 = t  3 = a  4 = b  5 = ext>
Evaluation of left hand side:
Specif of (local signal s in (p[s!/a] || q[s?/b])\s)
Behaviours:
    {<Bea u_p |0 4 5
         res |0 4 5>}--u_p--> local signal s in (x_p[s!/a] || q[s?/b])\s
    {<Bea u_q |0 3 5
         res |0 3 5>}--u_q--> local signal s in (p[s!/a] || x_q[s?/b])\s
    {<Bea u_p |3
         u_q |4
         res |0>}--tau--> local signal s in (x_p[s!/a] || x_q[s?/b])\s

Evaluation of right hand side:
Specif of (local atom t in (p[t/a]//t//q[t/b]) \\ t)
Behaviours:
    {<Bea u_p |3
         u_q |4
         res |0>}--tau--> local atom t in (x_p[t/a]//{t}//x_q[t/b]) \\ t
    {<Bea u_p |0 4 5
         res |0 4 5>}--u_p--> local atom t in (x_p[t/a]//{t}//q[t/b]) \\ t
    {<Bea u_p |0 3 5
         res |0 3 5>}--u_q--> local atom t in (p[t/a]//{t}//x_q[t/b]) \\ t
Proof succesfull:
" ⊢   Isbisim (local signal s in (p[s!/a] || q[s?/b])\s ==
          local atom t in (p[t/a] //{t} // q[t/b]) \\ t)"
```

Exemple 2.

Les propriétés de type commutativité ou associativité d'opérateurs dont les règles ne comportent pas de prédicats peuvent en général être prouvées en mode `logic`. Le parallèle MEIJE est commutatif:

```
@ logic;
Mode Logic Loaded.

@ parse par-comm = p // q == q // p;
Parsed par-comm : meije relation

@ prove par-comm;
Evaluation of left hand side:
Specif of (p // q)
    {p} {true} -- u_p --> x_p // q
    {q} {true} -- u_q --> p // x_q
    {p,q} {true} -- u_p.u_q --> x_p // x_q
```

```
Evaluation of right hand side:
Specif of (q // p)
    {q} {true} -- u_q --> x_q // p
    {p} {true} -- u_p --> q // x_p
    {p, q} {true} -- u_p.u_q --> x_q // x_p
Proof succesfull:
" ⊢   Isbisim (p // q == q // p)"
```

Il a suffit ici de retrouver quelle règle correspondait à quelle autre, et de vérifier que les résultats étaient bien dans la bisimulation. On a ainsi montré pour MEIJE un certain nombre de propriétés, parmi lesquelles:

$$
\begin{array}{rl}
\mathbb{0} \,//\, p = & p \\
p \,//\, q = & q \,//\, p \\
(p \,//\, q) \,//\, r = & p \,//\, (q \,//\, r) \\
\mathbb{0}[phi] = & \mathbb{0} \\
(u : p)[phi] = & u < phi > : p[phi]
\end{array}
\qquad
\begin{array}{rl}
p + \mathbb{0} = & p \\
p + q = & q + p \\
(p + q) + r = & p + (q + r) \\
p + p = & p
\end{array}
\qquad
\begin{array}{rl}
a * \mathbb{0} = & \mathbb{0} \\
a * b * p = & (a.b) * p \\
a * (b : p) = & (a.b) : a * p \\
\texttt{tau} * p = & p
\end{array}
$$

Exemple 3.

Le calcul *PLOTOS* [IsoLo] est une extension de *TCSP*, avec en particulier un nouvel opérateur *enable*.

```
@ display enable;
Operator enable :: process # process --> process
Semantics:
```

$$\texttt{enabling\_d} \quad \frac{p \xrightarrow{a} p' \;\&\; (a? = d)}{p >> q \xrightarrow{tau} q}$$

$$\texttt{enabling\_other} \quad \frac{p \xrightarrow{a} p' \;\&\; \texttt{not}\ (a? = d)}{p >> q \xrightarrow{a} p' >> q}$$

Nous montrons ici une tentative de simuler *enable* à l'intérieur même de *TCSP*, à l'aide d'un rendez-vous avec un synchronisateur. Cette simulation échoue, mais de peu: seul un comportement où l'action est tau différentie les spécifs.

Le synchronisateur est défini ici comme un opérateur supplémentaire que nous avons directement défini dans notre calcul à l'aide de ses règles de réécriture. C'est un petit automate qui, mis en rendez-vous avec un processus, va accepter n'importe quelle action, sauf l'action spéciale d, en restant dans le même état, ou accepter d et devenir l'horloge H_d. On utilise une action liée d', qui n'apparaitra donc pas dans les actions des processus formels. Elle ne peut ainsi être faite qu'une fois par $d' : q$. Donc le sous-terme $(p \,//\, \{a, b, d\} \,//\, H\_d)[d'/d]$ (obtenu après que p aît fait une fois l'action d) va se trouver bloqué par le rendez-vous extérieur sur d' dans le terme de droite de l'équation e ci-dessous, à l'exception d'actions *tau* éventuelles de p.

Nous voulons ici calculer dans un monde où l'alphabet est fini. Nous simulons cet alphabet fini par l'ensemble $\{a, b, d\}$, présent dans la contraintes sur p et q.

```
@ display synchro;
Operator synchro :: process
Semantics
synchro_d    ------------------
             synchro --d--> H_d
                    not (res ?= d)
synchro_else  ------------------
              synchro --res--> synchro
@ display H_d;
Operator H_d :: processus
Semantics
H    ------------
     H_d --d--> H_d

@ parse e = p >> q == local atom d' in ((p //{a,b,d}// synchro)[d'/d]  //{d'} //(d' : q)) \\ d';
Parsed e: tcsp equation
```

Une première tentative ayant mis en évidence le besoin d'un lemme, nous introduisons ici la relation contenant e et ce lemme.

```
@ parse lemme=q ==local atom d' in ((p //{a,b,d} // H_d)[d'/d]  //{d'} //(q)) \\ d';
Parsed lemme : tcsp equation

@ parse r = {e, lemme};
Parsed r: tcsp relation

@ prove r with   sorte {p,q} is atoms {a,b,d} signals none ;
Processing equation "p >> q ==
          local atom d' in ((p //{a,b,d} // synchro)[d'/d]  //{d'} //(d' : q)) \\ d'"
Base = <0 = tau 1 = d' 2 = a 3 = b 4 = d>
Evaluation of left hand side:
    {Bea <u_p |0 2 3
          res |0 2 3>} --u_p--> x_p >> q
    {Bea <u_p |4
          res |0>} --tau--> q

Evaluation of right hand side:
    {Bea <u_p |4                  % En rendez-vous sur d
          res |0>} --tau-->
          local atom d' in ((x_p //{a,b,d} // H_d)[d'/d]  //{d'} //(q)) \\ d';
    {Bea <u_p |2 3                % En rendez-vous sur a ou b
          res |2 3>} --u_p-->
          local atom d' in ((x_p //{a,b,d} // synchro)[d'/d]  //{d'} //(d' : q)) \\ d';
    {Bea <u_p |0                  % p tout seul
          res |0>} --tau-->
          local atom d' in ((x_p //{a,b,d} // synchro)[d'/d]  //{d'} //(d' : q)) \\ d';

Proof succeeded

Processing equation "q == local atom d' in ((p //{a,b,d} // H_d)[d'/d]  //{d'} //(q)) \\ d' "
Base = <0 = tau 1 = d' 2 = a 3 = b 4 = d>
Evaluation of left hand side:
    {Bea <u_q |0 2 3 4
          res |0 2 3 4>} --u_p--> x_q
```

```
Evaluation of right hand side:
   {Bea <u_q |0 2 3 4            % q tout seul
         res |0 2 3 4>} --u_q-->
         local atom d' in ((x_p //{a, b, d} // H_d)[d'/d]  //{d'} //(q)) \\ d';
   {Bea <u_p |0                  % p tout seul !!
         res |0>} --tau-->
         local atom d' in ((x_p //{a, b, d} // H_d)[d'/d]  //{d'} //(q)) \\ d';

Counter-example:
Rule 2 of right hand side has 1 free process p working,
and no rule of left hand side to match with.

Proof Negated:
" ⊢   Not-Isbisim {e, lemme}"
```

En effet p n'est pas complètement bloqué après avoir fait d. Il peut encore effectuer des actions internes, notre simulation n'est donc pas correcte. Pour obtenir de cette manière là une simulation de *enable*, il faudrait considérer une bisimulation modulo équivalence observationnelle (ignorer les *tau*-transitions). Vis à vis de la bisimulation forte, *enable* parait donc être un nouvel opérateur primitif.

Perspectives

Les développements prochains du système porteront d'abord sur l'introduction des simplificateurs. Dans un calcul donné, il devra être possible de mémoriser un ensemble de théorèmes déjà prouvés pour tenter d'en faire un système de réécriture complet canonique (soit à l'intérieur, soit à l'extérieur du système) à utiliser dans l'algorithme de bisimulation. De plus, cela permettra d'obtenir par réécriture des preuves plus directes de propriétés (conséquences inductives de nos théorèmes).

D'autre part, pour augmenter le domaine dans lequel on sait prouver des propriétés en mode "`logic`", il faudrait mettre en place un mécanisme de *simplification logique* des prédicats. Même un simplificateur de tautologies relativement simple permettrait certainement de conduire des preuves en l'absence des prédicats sophistiqués qui apparaissent dans les règles des opérateurs "synchronisants".

5. Conclusion

En l'état actuel du système on peut prouver en ECRINS des lemmes d'équivalence forte d'un genre relativement simple, en construisant des relations de bisimulation finies. Dans cette classe de problèmes se situent majoritairement les notions d'intertraductibilité de calculs, les lemmes principaux de simplification (par exemple *théorême d'expansion*. En *CCS*, comme dans les autres calculs qui n'utilisent pas de produit de simultanéïté, on peut traiter une classe autrement plus large de problèmes, dont toutes les équivalences entre termes (non clos) non récursifs. Certaines équivalence sonrt vérifiées universellement, d'autres moyennant certaines contraintes sur les *sortes* des variables de processus qui y apparaissent...

A plus long terme, et en dehors des améliorations futures déjà souhaitées dans le corps de l'article, on peut penser que tous les développements passés et à venir de la théorie devraient pouvoir s'incorporer dans ECRINS. Parmi ceux-ci on peut citer:

- l'introduction de structures d'actions plus variées, comme par exemple des *event structures* [Winskel].
- l'introduction de paramètres sur la nature de la bisimulation, bisimulations *observationnelles* bien sûr, mais aussi *critères d'abstraction* et *contextes*.

L'effort immédiat doit cependant maintenant porter sur le raffermissement des notions déjà existantes et de leur implémentation, ainsi que du confort d'utilisation du système par l'utilisateur. A ce prix ECRINS pourra apparaitre comme le premier système de manipulation générale de calculs de processus.

6. Bibliographie

[Au&Bo] D. Austry & G. Boudol, "Algèbre de processus et synchronisation", *Theoret. Comput. Sci. 30 p 91-131 (1984)*

[Boudol] G. Boudol, "Notes on algebraic calculi of processes", *Logics and Models of Concurrent Systems, NATO ASI Series F13, K. Apt, Ed.(1985)*

[BHR] S.D. Brookes, C.A.R. Hoare, A,W. Roscoe, "A Theory of Communicating Sequential Processes", *JACM Vol 31 (3) (1984)*

[GI] E. Gurari, O. Ibarra, "The Complexity of the Equivalence Problem for Counter Machines, Semilinear Sets and Simple Programs", *In: Proc. 11th Ann.ACM Symp on Theory of Computing (1979)*

[GS] S. Ginsburg, Spanier, "Bounded Algol-like languages", *Trans. Am. Math. Society 113 (1964)*

[He&Mil] M. Hennessy, R. Milner, "Algebraic Laws for Nondeterminism and Concurrency", *JACM 32 (1985)*

[HuetOpp] G. Huet, D. Oppen, "Equations and Rewrite Rules, a Survey", *in Formal Languages: Prespectives and Open Problems, Ed. R. Book, Academic Press (1980)*

[IsoLo] ISO, "Information Processing -Open Systems Interconnection- The definition of the specification language LOTOS" *Draft Proposal, ISO/DP 8807 97/21 N 423 (1985)*

[Larsen] K. Larsen, "Context Dependant Bisimulations", *PhD Thesis, University of Edinburg*, 1985.

[LCF] M. Gordon, R. Milner, C. Wadsworth, "Edinburgh LCF", *Springer-Verlag, LNCS no.78 (1979)*

[Mi0] R. Milner, "A Calculus of Communicating Systems", *Lectures Notes in Comput. Sci. 92 (1980)*

[Mi1] R. Milner, "Calculi for synchrony and asynchrony", *Theoret. Comput. Sci. 25 p267-310 (1983)*

[Mi2] R. Milner, "The Analysis of Concurrent Systems", *Lectures Notes in Comput. Sci. no.207 (1985)*

[Park] D. Park, "Concurrency and Automata on Infinite Sequences", *Lectures Notes in Comput. Sci. 104, p167-183 (1981)*

[Plotkin] G. Plotkin, "A structural approach to operationnal semantics", *Report Daimi FN-19, Comput. Sci. Dept., Aarhus Univ. (1981)*

[deS] R. de Simone, "Calculabilité et expressivité dans l'algèbre de processus Meije", *Thèse de troisième cycle, Université Paris 7 (1984)*

[RdS] R. de Simone, Higher-Level Synchronizing Devices in MEIJE*TCS 40 (1985)*

[Verga] D. Vergamini, "Verification by means of observational equivalence on automata", *Rapport Inria RR.501 (1986)*

[Winskel] G. Winskel, "Event Structures Semantics for *CCS* and Related Languages", *Daimi PB-159, Aarhus University (1983)*

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique